VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

POKROČILÉ METODY OPTIMALIZACE V KOMPILÁTORECH

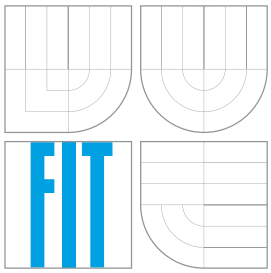
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAKUB MARTIŠKO

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

POKROČILÉ METODY OPTIMALIZACE V KOMPILÁTORECH

ADVANCED METHODS OF OPTIMIZATION IN COMPILERS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAKUB MARTIŠKO
AUTHOR

VEDOUCÍ PRÁCE — Prof. RNDr. ALEXANDER MEDUNA, CSc.
SUPERVISOR

BRNO 2013

Abstrakt

Tato práce se zabývá optimalizací zdrojového kódu při jeho překladu. Práce představuje některé v současnosti používané metody. Mimo to jsou zde zavedeny i metody nové, vycházející z vlastností booleovy algebry. Práce se pak také zabývá implementaci těchto metod.

Abstract

This thesis deals with optimization of source code during its compilation. The paper introduces some of the existing methods. The paper also introduces some new methods, that are based on properties of boolean algebra. Implementation of some of these methods is also described.

Klíčová slova

Kompilátory, optimalizace, logické výrazy

Keywords

Compilers, optimization, logical expressions

Citace

Jakub Martiško: Pokročilé metody optimalizace v kompilátorech, bakalářská práce, Brno, FIT VUT v Brně, 2013

Pokročilé metody optimalizace v kompilátorech

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Prof. RNDr. Alexandra Meduny, CSc. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jakub Martiško
14. května 2013

Poděkování

Rád bych poděkoval panu Prof. RNDr. Alexandru Medunovi, CSc. za trpělivé vedení práce a poskytnuté rady při její tvorbě. Dále bych rád poděkoval své rodině za podporu během mého studia.

Obsah

Kapitola 1

Úvod

Jednou z fází překladu zdrojového kódu na spustitelný program je jeho optimalizace. Jedná se o proces, který transformuje vstupní zdrojový kód takovým způsobem, že výstupní kód je proveden rychleji, potřebuje méně zdrojů nebo zabírá méně místa. Tato práce se zabývá prvním z těchto přístupů – zrychlením provedení programu. Na rozdíl od ostatních fází překladu, není optimalizace částí nutnou k vytvoření výsledného programu a je v rámci překladu dobrovolná. Jako taková pak předpokládá, že analyzovaný program je korektní, to například znamená, že neobsahuje neinicializované proměnné. V opačném případě, totiž může dojít ke změně zdrojového kódu i v místech, kde bychom to neočekávali a výstupní program pak může vykazovat prvky neplánovaného chování. Je důležité si uvědomit, že u jednotlivých optimalizací nesmí nikdy dojít ke stavu, kdy by se z korektně napsaného programu stal program nekorektní.

Literatura uvádí mnoho optimalizačních metod, které jsou mimo jiné zaměřeny na eliminaci opětovného vyhodnocování již jednou vypočtených výrazů, snížení počtu přístupů do paměti pomocí použití přímých hodnot namísto proměnných a zefektivňování cyklů. Optimalizace zaměřené specificky na logické operace jsou naproti tomu poměrně vzácné. Metody, které budou představeny v této práci, využívají vlastností booleovy algebry jako je idempotence, agresivita nuly a jedničky, neutralita nuly a jedničky a jiné.

Kapitola 2 slouží jako úvod do problematiky optimalizace zdrojového kódu. V této kapitole jsou stručně popsány některé z existujících optimalizačních metod. Dále jsou zde představeny a definovány základní pojmy sloužící k popisu podobných analýz. Následně je představen, a na jednoduchém příkladu demonstrován, společný framework, používaný k návrhu, implementaci a popisu jednotlivých optimalizací. Zároveň jsou zde i popsány vlastnosti algoritmů, vzniklých na základě tohoto frameworku.

Mnou navržené metody představuje kapitola 3. Jsou zde popsány čtyři hlavní metody optimalizace založené na vlastnostech booleovy algebry, včetně algoritmů pro jejich implementaci a vlastností těchto algoritmů. Tyto optimalizace pak určují hodnoty výrazů typu OR případně AND na základě konstantní hodnoty některého z jejich operandů, případně přeskupují jednotlivé výrazy a jejich podvýrazy takovým způsobem, aby se stal některý z podvýrazů mrtvým kódem. Dále pak například zkoumají, nenabývá-li výraz konstantní hodnoty a není-li tudíž tautologií či kontradikcí. U jednotlivých analýz jsou pak popsány vlastnosti, které určují, jak je teoreticky „kvalitní“ výsledek, který tyto metody poskytují.

Poslední kapitola 4 je věnována implementaci mnou zavedených analýz. Navržené metody jsem realizoval v jazyce Java za pomoci frameworku Soot. Pro ověření jejich funkčnosti jsem navrhl několik jednoduchých ukázkových příkladů, které je možné nalézt na přiloženém datovém nosiči.

Příloha B pak popisuje zprovoznění implementovaných metod. Dále je zde popsána struktura adresáře s implementací a výstupem jednotlivých optimalizací.

Kapitola 2

Principy optimalizace

Tato kapitola popisuje základní principy a metody užívané při optimalizaci zdrojového kódu. Nejprve budou stručně představeny některé z existujících optimalizačních metod. Mimo to je v této kapitole popsán také společný framework sloužící k popisu, analýze a návrhu metod, sloužících ke zkoumání a optimalizaci zdrojového kódu. Většina informací v této kapitole byla čerpána z [2], výjimku pak tvoří sekce 2.4, která vychází z [8].

2.1 Existující metody

2.1.1 Copy Propagation

Metoda se snaží nalézt takové výrazy, které přiřazují určité proměnné hodnotu uloženou v jiné proměnné. Touto proměnnou pak nahrazuje všechny následující výskyty této proměnné až do jejího předefinování. Cíl této optimalizace je znázorněn ukázkou 2.1. Tato optimalizace pak vytváří mrtvý kód (sekce 2.1.5).

$$T_1 = X \qquad\qquad T_1 = X$$
$$T_2 = T_1 + Y \qquad\qquad T_2 = X + Y$$

Obrázek 2.1: Kód před a po optimalizaci

2.1.2 Reaching Definitions

Tato analýza slouží ke zjištění místa, ve kterém byla proměnná, nalézající se v určitém bodě programu definována. Definicí proměnné se rozumí výraz, který této proměnné nastaví určitou hodnotu. Nenachází-li se mezi zkoumaným bodem P a bodem, ve kterém k této definici D došlo žádný výraz, který by přiřazoval této proměnné novou hodnotu, říkáme, že definice D *dosahuje* bodu P. [2]

Na základě znalosti místa definice dané proměnné pak můžeme rozpoznat neinicializované proměnné, případně určit, má-li proměnná v bodě P konstantní hodnotu. [2]

2.1.3 Analýza živých proměnných

Analýza živých proměnných (live variable analysis) testuje, zdali proměnná x v bodě p je použita v nějakém bodě, který může být dosažen s bodem p jako počátečním bodem. Pokud ano, proměnná je považována za *živou* v bodě p, pokud ne, nazývá se *mrtvou* v p. Tato analýza se často využívá k optimální alokaci procesorových registrů.

2.1.4 Available Expressions

Dle [2] je Výraz $a+b$ nazýván *dostupný* v bodě p, pokud neexistuje žádná cesta z počátečního bodu programu do bodu p, ve které by tento výraz nebyl vyhodnocen a pokud neexistuje žádné přiřazení do a nebo b mezi posledním z těchto vyhodnocení a bodem p. Pokud existuje (případně může existovat) přiřazení do některého z operandů a výraz není v tomto bloku po tomto přiřazení přepočítán, říkáme, že blok *zabíjí* výraz. Pokud je v bloku výraz určitě vypočítán a žádný z jeho operandů není následně změněn, říkáme, že blok *generuje* výraz. Tato analýza je použita jako jeden z kroků při optimalizaci *společných podvýrazů*.

2.1.5 Eliminace mrtvého kódu

Kód se nazývá živý, pokud je jím vypočítaná hodnota použita později během dalších výpočtů. Není-li výsledek určité skupiny výrazů dále nijak uplatněn, tvoří tato skupina mrtvý kód. Je poměrně nepravděpodobné že-by programátor vytvářel větší úseky mrtvého kódu cíleně. Častěji vzniká na základě ostatních optimalizačních metod. Jelikož další provádění programu není na něm nijak závislé, je možné tento kód ze zdrojového programu odstranit. Příkladem mrtvého kódu je výraz $T_1 = X$ ze sekce 2.1.1

2.1.6 Global Common Subexpression

Byla-li hodnota nějakého výrazu již spočítána, není nutné ji přepočítávat znovu, nalézá-li se tento výraz jako součást nějakého dalšího výpočtu. Takovéto výrazy jsou na nazývány *společné podvýrazy (common subexpression)*. Místo toho je výhodnější uložit si výsledek tohoto prvního výpočtu do pomocné proměnné a tu následně používat místo původního výrazu. Při předefinování některého z operandů tohoto výrazu je pak nutné celý výraz přepočítat a nadále používat již tuto novou verzi.

$$\begin{array}{ll} T_1 = X + Y & T_1 = X + Y \\ \dots & \dots \\ T_n = X + Y & T_n = T_1 \end{array}$$

Obrázek 2.2: Ukázka nahrazení společných podvýrazů

2.1.7 Constant Propagation

Cílem této metody je nahrazení takových výrazů, které jsou vždy vyhodnoceny jako konstantní hodnota právě touto hodnotou. Na rozdíl od dříve uvedených algoritmů, nesplňuje tento podmínku distributivity a jeho výsledkem tedy není MOP (viz sekce 2.3).

$$\begin{array}{ll} X = 2 & X = 2 \\ Y = 4 & Y = 4 \\ T_1 = X + Y & T_1 = 2 + 4 \\ T_2 = T_1 + Z & T_2 = 6 + Z \end{array}$$

Obrázek 2.3: Ukázka šíření konstant

2.2 Základní pojmy

V této části budou představeny základní pojmy, které budou později nutné k pochopení algoritmů pro optimalizační techniky popsané v této práci.

Definice 1. *Základní blok* je sekvence instrukcí, u kterých si můžeme být jistí, že budou provedeny přesně v určeném pořadí. Základní blok začíná:

- První instrukcí
- Navěštím skoku
- První instrukcí po instrukci skoku

Definice 2. *Stav programu* je množina hodnot každé proměnné v daném programu. Zpracování příkazu tedy provádí přechody mezi různými stavy programu. *Množina IN[S]* reprezentuje stav programu těsně před vykonáním příkazu S. Obdobně *množina OUT[S]* reprezentuje stav bezprostředně po provedení příkazu S.[2]

Definice 3. *Výpočetní cesta* z bodu p_1 do bodu p_n je posloupnost bodů $p_1, p_2, \ldots, p_n$ takových, že pro $i = 1, 2, \ldots, n-1$ platí jedno z následujících:

- p_{i-1} je bod bezprostředně předcházející příkaz p_i a p_{i+1} je bod bezprostředně následující tento příkaz
- p_i je poslední bod v daném bloku a p_{i+1} je první bod v bloku následujícím.

Obecně existuje nekonečně množství výpočetních cest a není tedy možné obecně analyzovat všechny možné varianty průběhu programu.

Definice 4. Vztah mezi hodnotami před a po vykonání příkazu popisuje *přechodová funkce*. [2] rozlišuje dva druhy přechodových funkcí:

- informace může být šířena dopředu, ve směru výpočtu programu – *forward flow*
- případně opačným způsobem, informace se šíří od konce programu směrem k jeho počátku – *backward flow*

Přechodová funkce pro příkaz s se značí jako $f(s)$. Pro forward flow problémy je vstupem přechodové funkce hodnota těsně před daným výrazem a výstupem pak hodnota bezprostředně po provedení tohoto výrazu. Tento vztah popisuje následující rovnice:

$$OUT[s] = f(IN[s]).$$

Obdobně pro backward flow problémy je vstupem funkce hodnota za zkoumaným výrazem a výstupem pak hodnota těsně před ním:

$$IN[s] = f(OUT[s]).$$

Pro popis celého bloku B sestávajícího z výrazů $s_1, s_2, \ldots, s_n$ kde s_1 je prvním výrazem tohoto bloku a s_n posledním platí, že $IN[B] = IN[s_1]$ a $OUT[s_n] = OUT[B]$. Složením přechodových funkcí pro jednotlivé výrazy dostaneme přechodovou funkci pro celý blok[2].

$$f_B = f_n \circ \cdots \circ f_2 \circ f_1$$

Obdobně jako pro jednotlivé výrazy pak zapisujeme $OUT[B] = f_B(IN[B])$ pro forward flow a $IN[B] = f_B(OUT[B])$ pro backward flow problémy.

Definice 5. Jelikož může mít základní blok více přímých předchůdců (například následník výrazu typu if-else), není obecně možné tvrdit $OUT[B_n] = IN[B_{n+1}]$. Místo toho se zavádí operátor *průseku (meet operator)* značený jako $\wedge$. $IN[B_{n+1}]$ se pak určí jako:

$$IN[B] = \bigwedge_{P \text{ kde P jsou jednotliví přímí předchůdci bloku B}} OUT[P].$$

Obdobný princip platí i pro backward flow problémy. Vstupní hodnota přechodové funkce je pak dána vztahem:

$$OUT[B] = \bigwedge_{S \text{ kde S jsou jednotliví přímí následníci bloku B}} IN[S].$$

2.3 Data-flow framework

Pro snadnější analýzu a použití jednotlivých optimalizačních metod se zavádí speciální framework. Díky němu je možné jednoduše určit vlastnosti jednotlivých algoritmů, zároveň také usnadňuje jejich návrh a implementaci. Framework pro data-flow analýzu $(D, V, \wedge, F)$ obsahuje následující části:

- Směr analýzy značený jako D.
- Polosvaz (více informací v sekci 2.3.1), skládající se z množiny V a operátoru $\wedge$.
- Množinu přechodových funkcí F. Funkce této množiny provádí zobrazení z V do V.

2.3.1 Polosvaz

Dle [2] je polosvaz[1] množina V a binární operátor průseku $\wedge$ přičemž platí $\forall x, y, z \in V$:

1. $$x \wedge x = x$$
2. $$x \wedge y = y \wedge x$$
3. $$x \wedge (y \wedge z) = (x \wedge y) \wedge z$$

Polosvaz má dva speciální prvky, *největší prvek* značený jako $\top$, který musí být součástí polosvazu a *nejmenší prvek* značený $\bot$, který není striktně vyžadován, které mají následující vlastnosti:

$$\forall x \in V; \top \wedge x = x$$

a pro nejmenší prvek:

$$\forall x \in V; \bot \wedge x = \bot.$$

> Jednotlivé prvky pak, v závislosti na použité analýze, reprezentují informaci o vlastnostech výrazů, proměnných a jiných prvcích programu pro všechny možné výpočetní cesty.[8]

[1] Svaz se od zde definovaného polosvazu liší existencí dvou operátorů. Zde popsaného průseku (nazývaného též infimum) a spojení (supremum, značeno jako $\vee$). Jelikož je v této práci využit pouze průsek a existencí spojení se nebudeme nijak zabývat, může docházet k záměně pojmů svaz a polosvaz. Vždy je však těmito pojmy myšlen polosvaz, tak jak je definován v sekci 2.3.1.

Pro reprezentaci dvouhodnotových vlastností jako je živost proměnné, platnost výrazu apod. se často používají jako prvky svazu *bitové vektory*. Pak například pro available expressions analýzu je vytvořen bitový vektor, jehož délka odpovídá počtu jednotlivých výrazů. Jednotlivé bity pak značí například, které z výrazů byly vygenerovány a jsou stále platné na konci daného základního bloku. Například pro základní blok popsaný na obrázku 2.4 by po provedení jednotlivých příkazů vypadal bitový vektor tak, jako popisuje tabulka 2.1 (nejlevější bit má index 2).

$T_1 := X + Y$
$T_2 := X + Z$
$T_3 := Y + Z$
$Y := 4$

Výraz	Odpovídající bit
$T_1 := X + Y$	0
$T_2 := X + Z$	1
$T_3 := Y + Z$	2

Obrázek 2.4: Základní blok reprezentovaný pomocí bitového vektoru

Analyzovaný příkaz	Bitový vektor
Žádný	000
$T_1 := X + Y$	001
$T_2 := X + Z$	011
$T_3 := Y + Z$	111
$Y := 4$	010

Tabulka 2.1: Odpovídající bitový vektor

2.3.2 Uspořádání

Předpokládejme polosvaz $(V, \wedge)$, *uspořádání* $\leq$ pak definujeme následovně:

$$\forall x, y \in V; x \leq y \text{ tehdy a jen tehdy když } x \wedge y = x.$$

Pár $(V, \wedge)$ nazýváme *(částečně) uspořádaná množina*. Díky vlastnostem průseku definovaným v sekci 2.3.1, pro toto uspořádání $\leq$ platí:[2]

1. $$x \leq x \text{ (reflexivita).}$$
2. $$\text{Pokud } x \leq y \text{ a zároveň } y \leq x, \text{ pak } x = y \text{ (antisymetrie).}$$
3. $$\text{Pokud } x \leq y \text{ a zároveň } y \leq z, \text{ pak } x \leq z \text{ (tranzitivita).}$$

2.3.3 Monotónnost a distributivita data-flow frameworku

Aplikujeme-li funkci $f \in F$ na dva libovolné prvky z V, takové že první není větší nežli druhý, a výsledné hodnoty těchto funkcí pak nejsou první větší nežli druhá, pak je tento framework monotónní. Alternativní definice[2] pak říká že, máme-li dvě hodnoty, na které

[2] Důkaz je možno nalézt v [2].

nejprve aplikujeme průsek a na tento výsledek aplikujeme přechodovou funkci, pak není tento výsledek větší než výsledek získaný aplikováním přechodových funkcí a teprve následným použitím průseku. Formálně lze tyto vztahy popsat rovnicemi 2.1 a 2.2.

$$\forall X, Y \in V \text{ a } f \in F, X \leq Y \rightarrow f(X) \leq f(Y) \tag{2.1}$$

$$\forall X, Y \in V \text{ a } f \in F, f(X \wedge Y) \leq f(X) \wedge f(Y) \tag{2.2}$$

Obdobnou vlastností je distributivita. Aby byl framework distributivní, musí platit že, máme-li dvě hodnoty, na které aplikujeme přechodovou funkci a na takto získané hodnoty následně operaci průseku získáme stejný výsledek jako bychom nejprve použili průsek na původní hodnoty a na tento výsledek teprve přechodovou funkci. Formálně popisuje tento vztah rovnice 2.3.

$$\forall X, Y \in V \text{ a } f \in F, f(X \wedge Y) = f(X) \wedge f(Y) \tag{2.3}$$

Platí že každý distributivní framework je také monotónní. Opačná podmínka ovšem neplatí a tudíž ne každý monotónní framework je zároveň i distributivní. Příkladem takového frameworku je constant propagation.

2.3.4 Iterativní algoritmus pro Data-flow problémy

Pro výpočet výsledných data-flow hodnot v množinách IN[B] a OUT[B] je dle [2] nutné znát následující údaje:

- Směr analýzy značený jako D
- Data-flow graf se speciálním *vstupním (entry)* a *výstupním (exit) uzlem*
- Množinu hodnot V, která mimo jiné obsahuje speciální konstantní hodnoty v_{entry} a v_{exit} reprezentující hodnoty vstupního a výstupního uzlu
- Operátor průseku $\wedge$
- Množinu funkcí F, kde $f_b \in F$ je přechodová funkce pro blok B

V závislosti na směru analýzy existují dvě základní verze algoritmu. Algoritmy 1 a 2 demonstruje obě tyto základní verze.

Algoritmus 1: Iterativní algoritmus pro forward flow problémy

$OUT[ENTRY] = v_{entry}$
for *každý blok B kromě bloku ENTRY* **do**
  $OUT[B] = \top$
end
while *Nastane změna některé z množin OUT* **do**
  for *každý blok B kromě bloku ENTRY* **do**
    $IN[B] = \bigwedge_{P \text{ kde P jsou jednotliví přímí předchůdci bloku B}} OUT[P]$
    $OUT[B] = f_B(IN[B])$
  end
end

Algoritmus 2: Iterativní algoritmus pro backward flow problémy

$IN[EXIT] = v_{exit}$
for *každý blok* B *kromě bloku* $EXIT$ **do**
  $IN[B] = \top$
end
while *Nastane změna některé z množin* IN **do**
  for *každý blok* B *kromě bloku* $EXIT$ **do**
    $OUT[B] = \bigwedge_{S \text{ kde S jsou jednotliví přímí následníci bloku B}} IN[S]$
    $IN[B] = f_B(OUT[B])$
  end
end

Díky vlastnostem data-flow frameworku popisuje [2] několik vlastností těchto algoritmů. Pro jednoduchost budeme předpokládat použití forward flow verze, ale vlastnosti zde popsané platí i pro opačný směr analýzy.

1. Pokud algoritmus 1 konverguje, pak jeho výsledek je řešením data-flow rovnic.
2. Pokud je příslušný framework monotónní, pak se nalezené řešení nazývá *maximum fixed point (MFP)*. MFP řešení je takové, pro které platí, že pro jakékoliv jiné řešení jsou hodnoty $IN[B]$ a $OUT[B] \leq$ (tak jak je tato operace definována v sekci 2.3.2) vzhledem k odpovídajícím hodnotám MFP.
3. Je-li svaz reprezentující framework monotónní a konečné výšky, pak je zaručeno, že algoritmus konverguje.

2.3.5 Vlastnosti Data-flow frameworku

V této sekci bude popsáno, jakého řešení dosáhneme v závislosti na vlastnostech daného frameworku. Pro zjednodušení popisu budeme opět předpokládat forward flow framework ale stejné (jen s přihlédnutím k významu množin IN a OUT) závěry platí pro oba směry analýzy. Všechny zde popsané postupy a důkazy pochází z kapitoly 9.3.4 knihy [2].

Definice 6. Pro nalezení *ideálního řešení* pro začátek určitého základního bloku B (značeno jako $IDEAL[B]$) je nejprve nutné nalézt všechny *možné* výpočetní cesty vedoucí z bodu $ENTRY$ do bodu B. Možná výpočetní cesta je taková, pro kterou existuje takový výpočet programu, který přesně tuto cestu využívá. Ideální řešení by tedy nalezlo všechny tyto cesty a na jim odpovídající výsledky přechodových funkcí by použilo průsek. Žádné spuštění programu by tudíž nemohlo vést k menším data-flow hodnotám než takto získaným. Formálně zapisujeme:

$$IDEAL[B] = \bigwedge_{P \text{ kde P jsou jednotlivé možné cesty z bodu ENTRY na začátek bloku B}} f_P(v_{entry}).$$

Vzhledem k operaci $\leq$ pro uspořádání definované pro tento framework platí:

- Jakékoliv řešení, které je větší nežli IDEAL je chybné.
- Jakékoliv menší řešení nežli IDEAL je konzervativní tj., bezpečné.

Řešení větší než IDEAL může vzniknout jen tak že opomeneme vypočítat určitou cestu. Nemůžeme si ovšem být jisti, že tato cesta neobsahovala určité příkazy, které by zamezily bezpečnosti prováděných analýz (například může v této cestě dojít k předefinování proměnné, ke kterému ve zbylých cestách nedojde). Naproti tomu menší řešení může vzniknout tak, že do výpočtu zahrneme i neexistující cesty, případně cesty, které nebudou během provádění programu nikdy využity. Tyto neexistující cesty mohou zavést nová omezení, která se v ideálním řešení nevyskytují. Výsledné řešení tedy nijak nemění korektnost programu, ale mohou existovat i další optimalizace, které toto řešení nepovolí.

Definice 7. Jelikož je určení všech možných cest obecně nerozhodnutelný problém, předpokládáme, že všechny cesty mohou být provedeny. *Meet-over-paths (MOP)* řešení tedy definujeme jako:

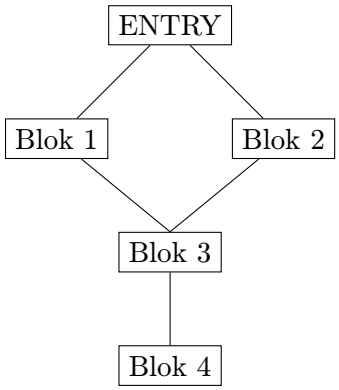
$$MOP[B] = \bigwedge_{P \text{ kde P jsou jednotlivé cesty z bodu ENTRY na začátek bloku B}} f_P(v_{entry}).$$

Jelikož je množina všech zkoumaných cest nadmnožinou obdobné množiny pro ideální řešení (jsou analyzovány i nevyužité cesty), je zřejmé že pro takto nalezené řešení platí $MOP[B] \leq IDEAL[B]$, zjednodušeně se pak zapisuje $MOP \leq IDEAL$.

V MOP řešení může být počet zkoumaných cest stále neomezený (například při zkoumání cyklů), není proto obecně možné vytvořit algoritmus pro výpočet MOP řešení. Iterativní algoritmus pracuje jiným způsobem, nesnaží se nejprve nalézt všechny cesty a ty následně analyzovat. Místo toho:

- Algoritmus navštěvuje jednotlivé bloky nezávisle na jejich pozici v rámci výpočtu
- Během každého setkání více bloků aplikuje průsek na všechna řešení v tomto bodě, přičemž některá z těchto řešení mohou být řešení bloků, které ještě nebyly analyzovány, a tudíž stále obsahují inicializační hodnotu.

Na rozdíl od MOP řešení, iterativní algoritmus se postupně snaží vypočítávat data-flow hodnoty pro jednotlivé bloky a pomocí průseku tuto výslednou hodnotu dále zpřesňovat. Pokusíme-li se analyzovat program popsaným obrázkem 2.5 a budeme-li se zajímat o hodnotu na začátku bloku 4.



Obrázek 2.5:

Pro MOP řešení získáme požadovaný výsledek pomocí vztahu 2.4.

$$MOP[B_4] = ((f_{B_3} \circ f_{B_1}) \wedge (f_{B_3} \circ f_{B_2}))(v_{entry}) \tag{2.4}$$

Výpočet při použití iterativního algoritmu, za předpokladu že jednotlivé bloky navštívíme v pořadí 1, 2, 3, 4 pak popisuje vztah 2.5. Takto získané řešení je označováno jako MFP (maximum fixed point).

$$MFP[B_4] = f_{B_3}((f_{B_1}(v_{entry})) \wedge (f_{B_2}(v_{entry}))) \tag{2.5}$$

Tyto dva výsledky jsou shodné pouze, je-li framework distributivní. Je-li framework pouze monotónní pak platí že $MFP[B_4] \leq MOP[B_4]$. Jelikož platí, že $MOP \leq IDEAL$ a zároveň $MFP \leq MOP$, pak také platí $MFP \leq IDEAL$.

2.3.6 Ukázka iterativního algoritmu

Pro demonstraci výše popsaného algoritmu použijeme *reaching definitions* analýzu 2.1.2. Cílem této analýzy je určit, kde byla proměnná v určitém bodě naposledy definována. Zavedeme speciální množiny *gen* a *kill*. Máme-li výraz tvaru $d_1 : \ X = Y + Z$ pak tento výraz generuje definici d_1 proměnné X a zabíjí všechny ostatní definice X nacházející se před tímto bodem. Přechodová funkce pro jednotlivé výrazy lze zapsat ve tvaru

$$OUT[s] = (IN[s] - kill_s) \cup gen_s.$$

Složením přechodových funkcí pro jednotlivé výrazy získáme přechodovou funkci pro celý blok B, kterou budeme zapisovat jako $f[B] = (IN[B] - kill_B) \cup gen_B$. Průsekem je v tomto případě množinové sjednocení. Jelikož hledáme nejbezpečnější řešení, je nutné nalézt pro jednotlivé body programu nejbližší předchozí definici proměnné. V případě že má daný blok více předchůdců, zajistí nám toto nalezení právě průnik. Program je prohledáván směrem od vstupního bloku směrem k výstupnímu, jedná se tedy o forward flow problém. Jednotlivé data-flow množiny jsou na začátku analýzy inicializovány jako prázdné, a během analýzy jsou do nich přidávány definice jednotlivých proměnných. Dosazením těchto hodnot do algoritmu 1 dostáváme algoritmus 3

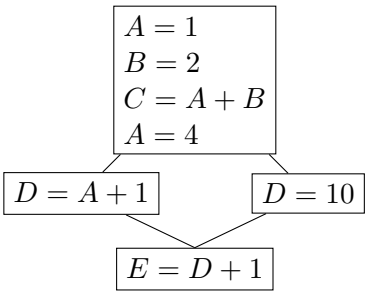
Algoritmus 3: Iterativní algoritmus pro reaching definitions analýzu

$OUT[ENTRY] = \emptyset$
for *každý blok B kromě bloku ENTRY* **do**
  $OUT[B] = \emptyset$
end
while *Nastane změna některé z množin OUT* **do**
  for *každý blok B kromě bloku ENTRY* **do**
    $IN[B] = \bigcup_{P \text{ kde P jsou jednotliví přímí předchůdci bloku B}} OUT[P]$
    $OUT[B] = (IN[B] - kill_B) \cup gen_B$
  end
end

2.4 SSA forma

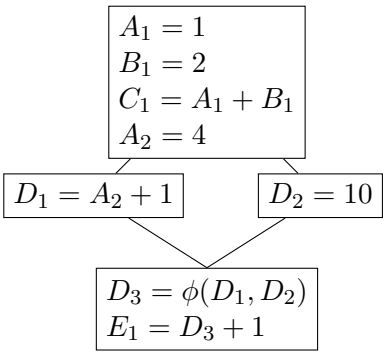
SSA forma (Static single assignment) je reprezentace mezikódu, která se vyznačuje existencí jediného přiřazení hodnoty jednotlivým proměnným. Kromě názvu proměnné je zavedena i její verze. Došlo-li by, v ne-SSA mezikódu, k předefinování proměnné, dojde v jeho SSA

verzi k inkrementaci čísla verze této proměnné a pro všechny její následující výskyty až do dalšího předefinování používá tuto verzi proměnné. Touto úpravou dojde ke zjednodušení mnoha optimalizací, jež musí kontrolovat, jsou-li údaje o jednotlivých proměnných stále platné. Existuje-li více přímých předchůdců bloku, ve kterých by došlo k přiřazení hodnoty proměnné se stejným názvem, zavádí se na začátek následujícího bloku speciální výraz nazývaný $\phi funkce$. Výsledná hodnota tohoto výrazu se pak ukládá do nové verze příslušné proměnné která je pak nadále využívána. $\phi funkce$ tedy slouží k určení, „aktuální“ verze proměnné.



Obrázek 2.6: Ukázkový kód

Obrázky 2.6 a 2.7 demonstrují stejný kód bez použití SSA formy a s jejím použitím.



Obrázek 2.7: Ukázkový kód v SSA formě

Kapitola 3

Optimalizace logických výrazů

Logické operátory, jejichž optimalizací se bude zabývat tato sekce, jsou AND (případně $\wedge$), OR ($\vee$) a NOT ($\neg$). Optimalizační metody představené v kapitole 2 je možno použít i pro optimalizaci logických výrazů. Mimo to, je ovšem možné využít i vlastností jednotlivých logických operací.

3.1 Vlastnosti logických výrazů

Jelikož je množina hodnot, kterých mohou jednotlivé operandy logických výrazů nabývat, dvouprvková, je často možné předvídat výslednou hodnotu celého výrazu. Mimo to, je také možné mnoho výrazů zjednodušit na základě vlastností booleovy algebry. V této práci je především využito následujících vztahů:

- Absorpce
$$x \vee (x \wedge y) = x \text{ a } x \wedge (x \vee y) = x$$
- Neutralita pravdivostních hodnot
$$x \vee 0 = x \text{ a } x \wedge 1 = x$$
- Agresivita pravdivostních hodnot
$$x \vee 1 = 1 \text{ a } x \wedge 0 = 0$$
- Komplementarita
$$x \vee \neg x = 1 \text{ a } x \wedge \neg x = 0$$
- Idempotence
$$x \vee x = x \text{ a } x \wedge x = x$$

3.2 Určení negací

Základní analýzou, která bude využita u dalších optimalizací, je určení negací jednotlivých proměnných. Každé proměnné x, je v každém bodě programu přiřazena množina $neg(x)$. Tato množina obsahuje seznam proměnných t takových že:

$$t = \neg x.$$

Obdobně je zavedena také množina $double\_neg(x)$, kde pro každou proměnnou $t_1 \in double\_neg(x)$ platí:

$$t_1 \in double\_neg(x) \iff t_1 = \neg t \text{ a zároveň } t \in neg(x).$$

Jinými slovy, tato množina obsahuje takové proměnné, které jsou negací negace nějaké jiné proměnné.

Obdobně jako v případě živých proměnných, je i zde nutné, při jakékoliv změně některé z proměnných, odpovídajícím způsobem změnit příslušné množiny.

Algoritmus sloužící k určení těchto množin prochází zdrojový kód ve směru provádění programu. Jednotlivé množiny jsou inicializovány jako prázdné. Přechodová funkce, zpracovávající jednotlivé výrazy je dána následujícím vztahem, který je vyhodnocován v tomto pořadí, a vždy jsou provedeny všechny splněné podmínky:

- Přiřazuje-li výraz hodnotu do proměnné x, pak je nejprve nutné nastavit množiny $neg(x)$ i $double\_neg(x)$ jako prázdné a zároveň odebrat proměnnou x ze všech množin, které ji obsahují.
- Je-li výraz tvaru $t_1 = \neg x$ pak $neg(x) = neg(x) \cup \{t_1\}$.
- Je-li výraz tvaru $t_2 = \neg t_1$ a zároveň platí $t_1 \in neg(x)$ pak $double\_neg(x) = double\_neg(x) \cup \{t_2\}$.
- Je-li výraz tvaru $t_3 = t_1$ a zároveň platí $t_1 \in neg(x)$ pak $neg(x) = neg(x) \cup \{t_3\}$ a obdobně pak pro množinu $double\_neg(x)$.

Vstupem jednotlivých základních bloků je pak průnik příslušných množin mezi jednotlivými přímými předchůdci tohoto bloku.

3.3 Vícenásobné negace

Jelikož není množství negací aplikovatelných na danou proměnnou teoreticky nijak omezeno (v praxi je ovšem použití více negací nepravděpodobné), je vhodné nahradit použití všech proměnných, které jsou obsaženy v některé z množin $double\_neg(x)$, touto proměnnou x. Tím dojde k zavedení mrtvého kódu a zároveň ke zjednodušení dalších optimalizací.

Díky tomu můžeme při popisu dalších metod předpokládat, že se ve vstupním programu již nenacházejí žádné vícenásobné negace.

3.4 Neutralita a agresivita pravdivostních hodnot

Známe-li hodnotu některého z operandů daného výrazu v daném bodě, můžeme pak poměrně jednoduše určit hodnotu celého výrazu. Při této optimalizaci využíváme neutrality, případně agresivity jednotlivých pravdivostních hodnot. Tuto optimalizační metodu je vhodné kombinovat s metodami typu constant propagation, které nahrazují proměnné mající v daném bodě konstantní hodnotu právě touto hodnotou.

Jakmile optimalizátor narazí na výraz tvaru X AND V nebo X OR V, kde X je proměnná a V přímá hodnota, může tento výraz nahradit výrazem dle tabulky 3.1.

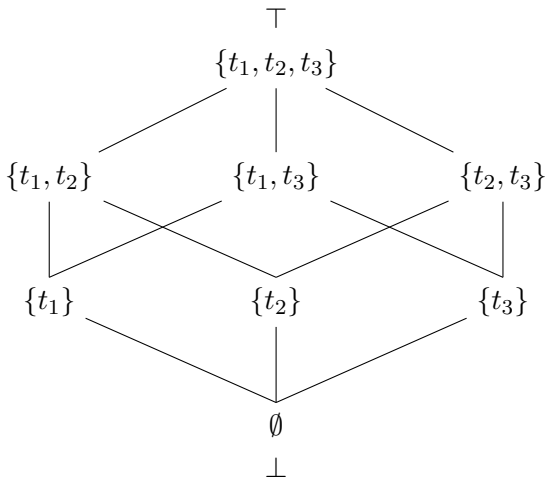
Opakovaným použitím, v kombinaci s metodou constant propagation je tento algoritmus schopen, i bez dalších úprav, zpracovávat i složitější podvýrazy. Tento postup je ovšem poměrně neefektivní a lze jej zrychlit. Jelikož nahrazení druhým operandem (řádky 2 a 3 v tabulce 3.1) se projeví jen v přímém následníkovi daného výrazu, budeme se zabývat jen

Operand 1	Operace	Operand 2	Výsledek
0	AND	X	0
0	OR	X	X
1	AND	X	X
1	OR	X	1

Tabulka 3.1: vztahy popisující neutralitu a agresivitu pravdivostních hodnot

nahrazením přímými hodnotami. Pro toto nahrazení slouží množiny *true_exp* a *false_exp*. Dojde-li k nahrazení některého z výrazů přímou hodnotou, je tento výraz přidán do příslušné množiny. Nachází-li se některý z operandů v jedné z těchto množin, je možné jej nahradit touto hodnotou a celý výraz případně nahradit přímou hodnotou a patřičně upravit množiny *true_exp* a *false_exp*. Dojde-li k přiřazení nové hodnoty výrazu, který se nachází v jedné z množin, je nutné je z této množiny odstranit a na základě této nové hodnoty přezkoumat jeho případné znovuzařazení. Jelikož tyto množiny určují, které výrazy nabývají konstantních hodnot, je nutné, aby daný výraz nabýval této hodnoty ve všech případných přímých předchůdcích daného bloku. Proto slouží jako operátor průseku množinový průnik.

Během inicializace je nastavena výstupní množina bloku ENTRY jako prázdná. Výstupní množiny všech ostatních bloků jsou pak inicializovány takovým způsobem, že obsahuje všechny možné proměnné daného programu. Polosvaz reprezentující tuto skupinu optimalizací demonstruje obrázek 3.1 (pro jednoduchost předpokládá jen existenci tří proměnných). Z tohoto obrázku je patrné, že výška svazu je konečná.



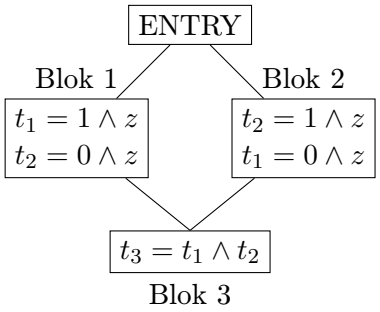
Obrázek 3.1: Polosvaz reprezentující příslušnost do množin true/false_exp

Jak ukazuje obrázek 3.2, nesplňuje tato skupina metod podmínku distributivity jak je popsána v sekci 2.3.3.

Označíme-li přechodové funkce pro bloky $B_1 \dots B_n$ jako $f_1 \dots f_n$ pak musí, pro program popsaný na obrázku 3.2, platit následující vztah:

$$f_3(f_1(ENTRY) \wedge f_2(ENTRY)) = f_3(f_1(ENTRY)) \wedge f_3(f_2(ENTRY)) \tag{3.1}$$

Zaměříme-li se nejprve na pravou stranu této rovnice, pak zjistíme, že každý z operandů



Obrázek 3.2: Distributivita

výrazu z bloku 3 se nalézá v množině *true_exp* v jednom z rodičů bloku 3. Budeme-li tedy analyzovat cestu $B_1 \rightarrow B_3$ pak bude výsledná množina (označme ji $true\_exp_{1,3}$) obsahovat prvky t_1 a t_3. Obdobně pak provedeme-li analýzu cesty $B_2 \rightarrow B_3$, výsledná množina bude mít tvar $true\_exp_{2,3} = \{t_2, t_3\}$. Aplikujeme-li na takto vzniklé množiny operaci průseku (v tomto případě se jedná o množinový průnik) získáme výslednou množinu obsahující pouze prvek t_3.

Naproti tomu pokusíme-li se nejprve spočítat průnik množin vzniklých jako výstupy přechodových funkcí f_1 a f_2 a takto vzniklou množinu využít jako vstup přechodové funkce f_3, zjistíme že, jelikož se jednotlivé *„pravdivé“* operandy nenacházejí oba ve stejném bloku, bude tato množina prázdná a funkce f_3 bude tudíž považovat jednotlivé operandy, výrazu z bloku 3, jako nekonstantní, a tudíž tento výraz nebude považovat za pravdivý. Tento výsledek ovšem neodpovídá výsledku, který jsme získali v předcházejícím odstavci, a tudíž levá strana rovnice 3.1 neodpovídá straně pravé a tento algoritmus tudíž není distributivní. Tabulka 3.2 demonstruje prvky obsažené v jednotlivých množinách při využití levé případně pravé strany rovnice 3.1.

Přechodová funkce	Výstupní množina *true_exp*
$f_1(ENTRY)$	$\{t_1\}$
$f_2(ENTRY)$	$\{t_2\}$
$f_3(f_1(ENTRY))$	$\{t_1, t_3\}$
$f_3(f_2(ENTRY))$	$\{t_2, t_3\}$
$f_3(f_1(ENTRY) \wedge f_2(ENTRY))$	$\emptyset$
$f_3(f_2(ENTRY)) \wedge f_3(f_1(ENTRY))$	$\{t_3\}$

Tabulka 3.2: Vyvrácení distributivity

Jelikož není algoritmus distributivní, je nutné dokázat, že splňuje alespoň podmínku monotónnosti, aby bylo zaručeno, že konverguje. Pro jednoduchost budeme předpokládat pouze vyhledávání výrazů, které jsou definitivně pravdivé. Důkaz pro definitivně nepravdivé výrazy je pak obdobný. Existuje pět (uvažujeme-li pouze logické operace a operaci přiřazení, ostatní druhy výrazů pak můžeme brát jako přiřazení konstantní hodnoty) základních případů, které mohou ovlivnit příslušnou množinu true_exp. Pro jednotlivé případy dokážeme, že provedené analýzy splňují podmínku monotónnosti.

1. Přiřazení přímé hodnoty proměnné
2. Výpočet výrazu typu OR, kde alespoň jeden z operandů je v množině true_exp

3. Výpočet výrazu typu OR, kde žádný z operandů není v množině true_exp
4. Výpočet výrazu typu AND, kde oba operandy jsou v množině true_exp
5. Výpočet výrazu typu AND, kde alespoň jeden z operandů není v množině true_exp

První z těchto případů, přiřazení konstanty, se dále dělí na dva typy. Prvním z nich je přiřazení pravdivé hodnoty, druhým pak přiřazení jakékoliv jiné hodnoty (nepravda, proměnná, která není v množině true_exp, neznámý výsledek aritmetické operace...). Pro přiřazení pravdivé hodnoty platí, že je-li cílová proměnná již v množině true_exp, přechodová funkce je pak pouze funkcí identity, v opačném případě pak dojde v rámci příslušného svazu k posunu o jeden prvek směrem k prvku $\top$. Při přiřazování jiných hodnot pak obdobně platí, že pokud není proměnná v množině true_exp, jedná se o funkci identity, v opačném případě dojde k posunu o jeden prvek směrem k $\bot$. Druhý, z výše vyjmenovaných případů, cílovou množinu true_exp nijak nemění v případě, že cílová proměnná již je členem této množiny. V opačném případě dojde k posunu o jeden prvek směrem k prvku $\top$. Třetí z možnosti se chová opačným způsobem nežli možnost druhá, to znamená, že pokud je cílová proměnná v množině true_exp, je následně z této množiny odstraněna, v opačném případě pak opět nedochází k žádné změně. Případy 4 a 5 se chovají podobným způsobem jako případy 2 respektive 3. Z tohoto popisu vyplývá, že kdykoliv může dojít ke změně nejvýše o jednu úroveň v rámci příslušného svazu. Tyto vztahy popisuje tabulka 3.3. Jelikož dochází vždy k posunu

Řádek	Tvar výrazu	Cílová proměnná	Operand 1	Operand 2	Směr posunu
1	$t_1 = x$	$t_1 \in true\_exp$	$x \in true\_exp$		—
2	$t_1 = x$	$t_1 \notin true\_exp$	$x \in true\_exp$		$\uparrow$
3	$t_1 = x$	$t_1 \in true\_exp$	$x \notin true\_exp$		$\downarrow$
4	$t_1 = x$	$t_1 \notin true\_exp$	$x \notin true\_exp$		—
5	$t_1 = x \vee y$	$t_1 \in true\_exp$	$x \in true\_exp$	$y \in true\_exp$	—
6	$t_1 = x \vee y$	$t_1 \notin true\_exp$	$x \in true\_exp$	$y \in true\_exp$	$\uparrow$
7	$t_1 = x \vee y$	$t_1 \in true\_exp$	$x \notin true\_exp$	$y \in true\_exp$	—
8	$t_1 = x \vee y$	$t_1 \notin true\_exp$	$x \notin true\_exp$	$y \in true\_exp$	$\uparrow$
9	$t_1 = x \vee y$	$t_1 \in true\_exp$	$x \notin true\_exp$	$y \notin true\_exp$	$\downarrow$
10	$t_1 = x \vee y$	$t_1 \notin true\_exp$	$x \notin true\_exp$	$y \notin true\_exp$	—
11	$t_1 = x \wedge y$	$t_1 \in true\_exp$	$x \in true\_exp$	$y \in true\_exp$	—
12	$t_1 = x \wedge y$	$t_1 \notin true\_exp$	$x \in true\_exp$	$y \in true\_exp$	$\uparrow$
13	$t_1 = x \wedge y$	$t_1 \in true\_exp$	$x \notin true\_exp$	$y \in true\_exp$	$\downarrow$
14	$t_1 = x \wedge y$	$t_1 \notin true\_exp$	$x \notin true\_exp$	$y \in true\_exp$	—
15	$t_1 = x \wedge y$	$t_1 \in true\_exp$	$x \notin true\_exp$	$y \notin true\_exp$	$\downarrow$
16	$t_1 = x \wedge y$	$t_1 \notin true\_exp$	$x \notin true\_exp$	$y \notin true\_exp$	—

Tabulka 3.3: Směr posunu v rámci svazu

nejvýše o jeden prvek, není možné, aby pro dvě vstupní množiny $true\_exp_1$ a $true\_exp_2$, pro které platí že $true\_exp_1 < true\_exp_2$ a zároveň $f(true\_exp_2) = true\_exp_2$ došlo k situaci kdy $f(true\_exp_1) > f(true\_exp_2)$. Tato vlastnost dokazuje monotónnost pro všechny dvojice vstupních hodnot kdy dochází ke změně příslušné množiny jen pro jeden případ z této dvojice. Dochází-li k posunu ve stejném směru v obou případech, pak je zřejmé, že, díky posunu o maximálně jednu úroveň, nebude monotónnost opět porušena. Nyní zbývá

prozkoumat případy, kdy dochází k posunu v opačném směru (např. řádky 8 a 9 tabulky 3.3). Jak ukazuje tabulka pro řádky 8 a 9 ve výchozím stavu je pro obě tyto možnosti v příslušných množinách true_exp stejný počet prvků (konkrétně 1). Z toho vyplývá, že pro množiny před provedením přechodových funkcí platí (z hlediska porovnávacích operací v rámci uspořádaní) že $true\_exp_1 = true\_exp_2$. Není tedy možné, aby došlo v rámci uspořádání, k „prohození“ těchto množin. Pro kombinaci řádků 6 a 9 platí že, množina odpovídající řádku 6 se nachází blíže prvku $\top$ a množina pro řádek 9 blíže prvku $\bot$. Jelikož po provedení přechodových funkcí se tyto množiny ještě více přiblíží ve směru těchto mezních hodnot, platí i příslušná podmínka monotónnosti. Obdobný postup lze použít i pro řádky 11 až 16 tabulky 3.3.

3.5 Komplementarita a idempotence pravdivostních hodnot

Podobný přístup jako 3.4 je využití komplementarity a idempotence logických operací. Oproti předchozí metodě má tu výhodu, že nemusíme předem znát hodnoty jednotlivých operandů. Tato metoda využívá množinu $neg(x)$ představenou v sekci 3.2.

Algoritmus vyhledává výrazy tvaru X AND Y nebo X OR Y kde $Y = X$ nebo $Y \in neg(X)$ a nahrazuje je podle tabulky 3.4.

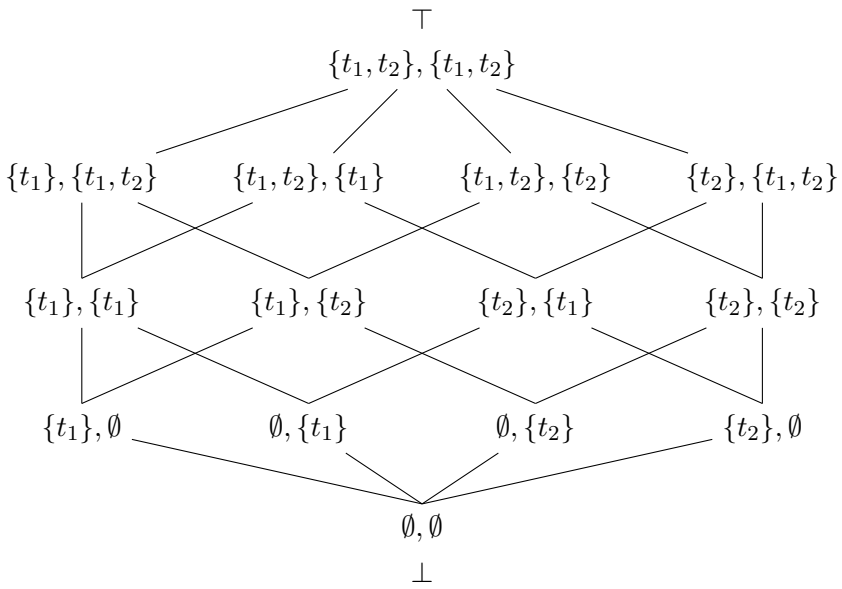
Operand 1	Operace	Operand 2	Výsledek
X	AND	X	X
X	OR	X	X
X	AND	neg(X)	0
X	OR	neg(X)	1

Tabulka 3.4: vztahy popisující komplementaritu

Pro optimalizaci složitějších výrazů je nutné tyto metody modifikovat takovým způsobem, aby algoritmus dokázal zpracovat i příkazy, které vyhodnocují jednotlivé podvýrazy daného výrazu a vhodně je přeskupit. Cílem tohoto přeskupení je prozkoumat příslušnou posloupnost instrukcí a na základě komutativity se pokusit najít páry proměnných a jejich negací. Pokud takovýto pár nalezne, může nahradit příslušný výraz přímou hodnotou. Podvýrazy, které původně vedli k výpočtu tohoto výrazu, se pak mohou stát mrtvým kódem.

Algoritmus zajišťující toto přeskupení prochází program ve směru zpracování kódu (forward flow problém). Při prvním průchodu si pro každý bod programu a každý výraz sestaví množinu *variables[exp]*, která obsahuje názvy proměnných (společně s případným příznakem členství v nějaké množině neg), se kterými tento výraz pracuje. Narazí-li optimalizátor na instrukci, která přiřazuje novou hodnotu proměnné, která se již nachází v alespoň jedné množině *variables[exp]*, je nutné ji ze všech těchto množin odstranit. Má-li základní blok více přímých předchůdců, ve kterých se mohou vyskytovat některé z podvýrazů zkoumaného výrazu, pak je jako výsledná množina *variables[]* použit průnik příslušných množin v jednotlivých blocích. Následně prochází algoritmus program znovu, a jakmile narazí na příkaz s logickou operací, zjistí, jakého druhu jsou jeho operandy. Je-li operand výsledná hodnota nějakého předchozího logického výrazu, který využívá *stejnou* operaci (konjunkce případně disjunkce) jako momentálně zkoumaný výraz, pak algoritmus prozkoumá příslušnou množinu *variables[exp]*. Jakmile takto zpracuje oba své operandy, pokusí se najít dvojici X a Y kde $Y = \neg X$ a $X \in variables[operand1]$ a zároveň $Y \in variables[operand2]$.

Nalezne-li algoritmus tuto dvojici, je pak možné nahradit celý tento výraz hodnotou $TRUE$ případně $FALSE$ v závislosti na použité operaci. Dojde-li k tomuto nahrazení, je vhodné následně provést analýzu na živost jednotlivých podvýrazů.



Obrázek 3.3: Polosvaz reprezentující příslušnost do množin variables[]

Obdobně jako u předchozí skupiny metod 3.4 jsou jednotlivé množiny inicializovány takovým způsobem, aby obsahovali všechny prvky daného univerza. Výjimkou je pak opět skupina množin reprezentující stav na konci vstupního bloku *ENTRY*, tyto množiny jsou nastaveny jako prázdné. Příslušný polosvaz demonstruje obrázek 3.3. Tento ukázkový svaz předpokládá existenci dvou výrazů a tedy i dvou množin variables pro jednotlivé body programu. Dále předpokládá existencí dvou proměnných. Z obrázku je patrné, že výška svazu je pro konečný počet proměnných a výrazů konečná.

Přechodové funkce těchto optimalizací lze popsat pomocí takzvané *gen-kill[2]* formy. Gen-kill forma zavádí dvě speciální množiny pro popis činnosti přechodových funkcí. Množina *gen* obsahuje všechny prvky, které v rámci zkoumané jednotky (základní blok případně jednotlivé výrazy) jsou shledány, že splňují analyzovanou vlastnost. V případě této skupiny optimalizací se tedy jedná o množinu definovaných výrazů. Naproti tomu množina *kill* obsahuje ty prvky, jejichž platnost aktuální zkoumaná jednotka ruší, v tomto případě tedy ty proměnné, u kterých došlo k přiřazení nějaké nové hodnoty (průnik množin gen a kill nemusí být prázdný). Výsledná data-flow hodnota je tedy pro daný blok (případně samostatný výraz) vypočtena ze vstupní hodnoty $IN[B]$ vztahem 3.2.

$$OUT[B] = gen \cup (IN[B] - kill) \tag{3.2}$$

Rovnice popisující distributivitu tohoto frameworku má tedy tvar:[1]

$$(gen \cup (IN_1 - kill)) \cap (gen \cup (IN_2 - kill)) = gen \cup ((IN_1 \cap IN_2) - kill). \tag{3.3}$$

Při vyhledávání nových proměnných, které by mohly být přidány do některé z množin *variable*[] nás nijak nezajímá aktuální stav těchto množin. Zkoumáme jen tvar jednotlivých

[1] Zde popsaný důkaz vychází z velmi podobného důkazu z kapitoly 9.3.2 knihy [2]

analyzovaných výrazů nezávisle na jejich jednotlivých proměnných, případně typu těchto proměnných. Množina gen bude tedy mít v rámci základního bloku vždy stejnou hodnotu nezávisle na pořadí zpracování tohoto bloku. Jelikož není množina gen nijak závislá na hodnotách z předchozích bloků, je zřejmé, že všechny výskyty této množiny na levé i pravé straně rovnice 3.3 nabývají stejné hodnoty a je možné je tedy z rovnice vypustit. Zkoumaná rovnici je tedy možné zjednodušit na tvar popsaný vztahem 3.4.

$$(IN_1 - kill) \cap (IN_2 - kill) = (IN_1 \cap IN_2) - kill \tag{3.4}$$

Obdobně jako u množiny gen, tak i množina kill obsahuje stejné prvky nezávisle na pořadí zpracování daného bloku. Pro sestavení této množiny stačí vytvořit seznam všech proměnných, které byly v rámci tohoto bloku předefinovány, je zřejmé, že tato operace produkuje stejné výsledky nezávisle na zpracování ostatních bloků. Z vlastností množinových operací (str. 75 [3]) vyplývá, že levá strana rovnice 3.4 je ekvivalentní straně pravé. Čímž je dokázána distributivita tohoto frameworku a tudíž řešením tohoto algoritmu je MOP řešení. Jelikož je framework distributivní je také splněna podmínka monotónnosti a tudíž i konvergence.

3.6 Absorpce

Tato optimalizace je provedena ve dvou fázích. V první jsou sestaveny pro každý bod programu množiny všech logických výrazů, které jsou dostupné v daném bodě. Ve druhé fázi jsou pak jednotlivé instrukce zdrojového programu nahrazovány v závislosti na těchto množinách.

Sestavení množin obstarává algoritmus vycházející z *available expressions* analýzy. Pro jednotlivé body programu jsou vytvořeny množiny *and_exp* a *or_exp*. Program je poté procházen ve směru zpracování programu a dle použité operace jsou do množin *and_exp* a *or_exp* ukládány trojice obsahující operandy a cílové proměnné jednotlivých výrazů. Je-li některý z operátorů negací nějaké jiné proměnné, je vhodné tuto skutečnost také zaznamenat, aby nebylo nutné ve druhé fázi kontrolovat také množiny *neg* ze sekce 3.2. Množiny vstupující do základního bloku vzniknou jako průnik příslušných množin ve všech přímých předchůdcích tohoto bloku. Množiny jsou na začátku inicializovány na množinu obsahující všechny prvky daného univerza. Výjimku pak tvoří vstupní blok, který je inicializován jakožto prázdná množina.

Přechodové funkce popisující sestavení množin *and_exp* a *or_exp* lze stejně jako v případě analýzy 3.5 zapsat v podobě gen-kill formy. Obdobným způsobem jako v případě 3.5 lze tedy i dokázat distributivitu tohoto frameworku.

$E_1 : \ t_1 = \neg X \wedge Y$

$E_2 : \ t_2 = X \vee Y$

$E_3 : \ t_1 = X \wedge Y$

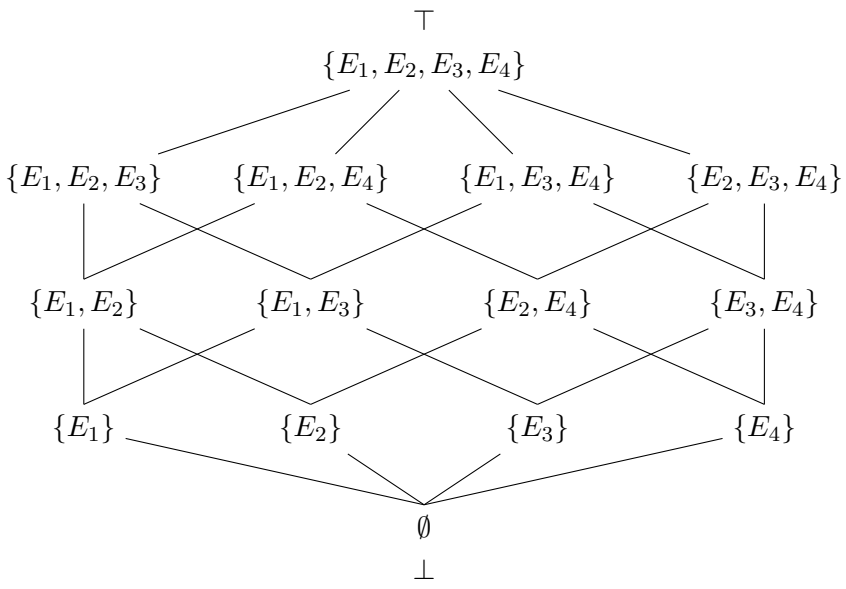
$E_4 : \ t_4 = t_1 \vee X$

Zpracované výrazy	Množina *and_exp*	Množina *or_exp*
Žádný	$\emptyset$	$\emptyset$
E_1	$\{E_1\}$	$\emptyset$
E_2	$\{E_1\}$	$\{E_2\}$
E_3	$\{E_3\}$	$\{E_2\}$
E_4	$\{E_3\}$	$\{E_2, E_4\}$

Obrázek 3.4: Příklad demonstrující optimalizaci založenou na absorpci.

Výsledek první fáze analýzy programu je demonstrován tabulkou a jednoduchým ukázkovým programem na obrázku 3.4

Jak ukazuje obrázek 3.5, je výška příslušných data-flow svazů konečná. Společně s distributivitou tohoto frameworku to tedy znamená, že výsledné řešení zde popsaného algoritmu je meet over paths řešením.



Obrázek 3.5: Svaz reprezentující možné data-flow hodnoty množiny *and_exp* pro výrazy definované v programu 3.4

Ve druhé fázi vyhledává analyzátor logické výrazy a zkoumá jejich tvar dle tabulky 3.5. K určení tvaru výrazu T_1 slouží množiny *and_exp* a *or_exp*, vytvořené v první části analýzy. Nalezne-li algoritmus vyhovující tvar, nahradí výraz T_2 příslušnou výslednou hodnotou. Výraz T_1 se pak může stát mrtvým kódem.

Výraz T_2	Operand T_1	Výsledná hodnota
$X \vee T_1$	$X \wedge Y$	X
$X \wedge T_1$	$X \vee Y$	X
$X \vee T_1$	$\neg X \wedge Y$	$X \vee Y$
$X \wedge T_1$	$\neg X \vee Y$	$X \wedge Y$

Tabulka 3.5: Absorpce

3.7 Tabulka

Tato skupina metod se snaží nejprve sestavit tabulku s jednotlivými logickými výrazy pro základní blok. Následně zkoumá jakým způsobem je ovlivněna hodnota výrazu v závislosti na jednotlivých operandech. Na základě těchto optimalizací je možné určit, je-li výraz (případně jeho část) tautologií případně kontradikcí.

Tabulky používané těmito metodami se skládají ze dvou část. První část obsahuje seznam všech proměnných pro některý z výrazů. Druhá část pak jednotlivé podvýrazy, včetně cílové proměnné pro uložení výsledku, názvů jednotlivých operandů a operaci použitou v tomto výrazu. Pro každou sérii podvýrazů, která celkově tvoří jeden složený výraz je

vytvořena samostatná tabulka. Po sestavení, jsou jednotlivé proměnné v tabulkách nastaveny na všechny možné kombinace hodnot a na jejich základě jsou vypočteny hodnoty jednotlivých výrazů a ty jsou následně zkoumány.

3.7.1 Sestavení tabulky

Algoritmus stojící za sestavením tabulky prochází kód programu proti směru jeho zpracování. Narazí-li na logický výraz, pokusí se jej najít v již existujících tabulkách. Pokud jej nenalezne, pak vytvoří tabulku novou a do části pro proměnné uloží jeho operandy. Do části pro výrazy pak samotný výraz včetně jména proměnné pro uložení výsledku. Pokud tento výraz v některé z tabulek již je, pak, je-li v sekci pro proměnné, je přesunut do sekce pro výrazy a jeho operandy jsou uloženy do sekce pro proměnné. Je-li některý z jeho operandů již v sekci pro výrazy, je nutné všechny již existující výskyty této proměnné v tabulce přejmenovat (při použití SSA formy tento problém zaniká) a do tabulky uložit tento nový výskyt jakožto novou proměnnou. Je-li v sekci pro výrazy, pak se jedná a znovuvýpočet stejné hodnoty (což lze eliminovat pomocí metod zaměřených na společné podvýrazy), nebo došlo ke změně hodnoty některého z operandů a původní verze tohoto výrazu se již v tabulce nachází pod novým jménem. Nalezne-li algoritmus příkaz přiřazující konkrétní hodnotu nějaké proměnné, pak je tato hodnota nastavena jako konstantní pro všechny výskyty této proměnné a proměnná je přejmenována ve všech tabulkách (při použití SSA nutnost přejmenování odpadá).

Jakmile jsou jednotlivé tabulky sestaveny, doplní algoritmus hodnoty jednotlivých operandů na všechny možné kombinace. Na základě těchto vstupních hodnot pak analyzátor vypočte výsledné hodnoty jednotlivých výrazů a doplní je do tabulek. Takto připravené tabulky jsou pak dále zkoumány.

3.7.2 Analýza tabulek

Základní optimalizací, kterou tyto metody provádějí, je ověření nenabývá-li daný výraz konstantní hodnoty nezávisle na vstupních parametrech. Tato optimalizace spočívá v prostém prozkoumání výsledků jednotlivých výrazů a zkontrolování nemají-li výsledky pro všechny možné vstupní hodnoty identickou hodnotu. Takový podvýraz je pak tautologií případně kontradikcí a je tedy možné jej nahradit příslušnou konstantou. Takto se mohou stát veškeré výrazy, které vedou k výpočtu tohoto výrazu, mrtvým kódem. Jelikož testujeme všechny možné kombinace vstupních hodnot, je zřejmé že tato transformace neporuší korektnost kódu.

Další možností jak tyto výrazy analyzovat je vzájemné porovnání jednotlivých sloupců tabulky. Jsou-li dva sloupce stejné, jinými slovy nabývají-li dva výrazy při stejných vstupech vždy shodné hodnoty, je možné nahradit jeden z těchto výrazů výrazem druhým.

$$t_1 = x + y$$

$$t_2 = v + w$$

$$t_3 = t_1 + t_2$$

Obrázek 3.6: Analyzovaný kód

Pokusíme-li se takto analyzovat ukázkový kód 3.6, pak zjistíme, že dva výrazy[2] s různými operandy nemohou mít nikdy shodné všechny řádky (zanedbáme-li případ tautologie případně kontradikce), jelikož se v tabulce nacházejí všechny operandy, které jsou součásti alespoň jednoho podvýrazu z celého zkoumaného výrazu. To znamená, že aby měly dva výrazy vždy stejnou hodnotu, musí pracovat se stejnou množinou operandů, v opačném případě se totiž budou lišit právě v řádcích, které mění hodnoty proměnných, které jsou pro tyto výrazy unikátní. Pracují-li oba výrazy se stejnou množinou operandů, pak díky pravdivostním hodnotám a vlastnostem operací AND a OR je zajištěna bezpečnost této transformace. Příkladem této transformace je tabulka 3.6, která popisuje výraz daný rovnicí 3.5.

$$T_3 = X \wedge ((Y \vee Z) \wedge X) \tag{3.5}$$

X	Y	Z	$T_1 = Y \vee Z$	$T_2 = T_1 \wedge X$	$T_3 = T_2 \wedge X$
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

Tabulka 3.6: Tabulka reprezentující výraz popsaný rovnicí 3.5

Z tohoto příkladu je patrné že výraz T_3 se oproti výrazu T_2 již dále nemění.

[2]znak + zde nahrazuje některou z operací AND případně OR

Kapitola 4

Implementace

Pro implementaci mnou navržených funkcí jsem použil Java framework *Soot*[1]. Tento framework je navržen k provádění data-flow analýz a optimalizací kódu napsaném v jazyce Java. Framework je volně dostupný z oficiálních stránek projektu [1] a v době psaní této práce je licencován jako GNU LGPL. Tento framework obsahuje implementaci mnoha často používaných optimalizací, mimo to ovšem umožňuje i vytvoření a použití implementací vlastní. Pro ověření funkčnosti navržených metod jsem ovšem existující žádné analýzy nevyužil. Pro implementaci vlastních optimalizačních metod slouží třídy reprezentující algoritmus popsaný v sekci 2.3.4. K dispozici je verze jak pro forward (*ForwardFlowAnalysis*), tak i pro backward(*BackwardFlowAnalysis*) flow verzi. Mimo to existují i speciální *Branched* verze těchto metod, lišící se tím, že v případě větvení programu mohou předávat jednotlivým větvím rozdílné hodnoty. Framework dokáže reprezentovat optimalizovaný zdrojový kód pomocí několika různých mezikódů. Při implementaci vlastních analýz jsem využil reprezentace nazvané *Jimple*. Jedná se o tříadresný kód, s malou množinou jednotlivých druhů příkazů[5]. Mimo to existuje i SSA varianta Jimple reprezentace nazvaná *Shimple*, kterou jsem se nakonec rozhodl nevyužít, abych tím demonstroval použitelnost navržených analýz i pro ne-SSA kód. Jednotlivé zkoumané prvky jsou reprezentovány pomocí datového typu *Unit*. Tento datový typ slouží jako jistý „kontejner“ pro jednotlivé instrukce[6]. Datovým typem sloužícím k reprezentaci jednotlivých výrazů je takzvaný *ValueBox*[1]. Jednotlivé proměnné typu Unit pak obsahují seznamy „definičních“ ValueBoxů, které reprezentují levou stranu příkazu, „výpočetních“ ValueBoxů, sloužících jako pravá strana příkazu a případné odkazy na jiné jednotky Unit[6].

Nezávisle na použitém směru analýzy, je, kromě konstruktoru této třídy, nutné implementovat také následující metody[7][5]:

- *void flowThrough(Object in, Unit d, Object out)*[2]
- *Object newInitialFlow()*
- *Object entryInitialFlow()*
- *void merge(Object in1, Object in2, Object out)*
- *void copy(Object source, Object dest)*

[1] Jedná o ukazatel na konkrétní hodnotu, která reprezentuje výraz[6]

[2] Pro branched verzi analýz má tato metoda signaturu *void flowThrough(Object in, Unit d, List<Object>fallOut, List<Object>branchOuts)*, kde seznam *fallOut* obsahuje hodnoty pro pravdivou větev a seznam *branchOuts* pro nepravdivé větve.

Metody *newInitialFlow* a *entryInitialFlow* slouží k inicializaci jednotlivých prvků data-flow grafu. Metoda entryInitialFlow nastavuje data-flow hodnotu vstupního prvku (ENTRY případně EXIT v závislosti na směru), newInitialFlow pak hodnoty všech ostatních prvků.

Pro šíření data-flow hodnot mezi jednotlivými jednotkami slouží metody *merge* a *copy*. O předání výstupní hodnoty zkoumané jednotky na vstup jednotky následující se stará funkce *copy*, šíření hodnoty při existencí více přímých předchůdců (následníku) a tudíž implementace operace průseků je provedena v metodě *merge*.

Samotnou přechodovou funkci pro jednotlivé základní bloky (v tomto případě reprezentovány pomocí datového typu Unit) pak popisuje metoda *flowThrough*. V této metodě tedy dochází k transformaci vstupních data-flow hodnot zkoumaného bloku na hodnoty výstupní. V případě branchedFlow analýz pak může obsahovat více výstupních hodnot v závislosti na větvení programu.

4.1 Implementace vlastních metod

V této sekci bude popsána implementace některých z metod, popsaných v této práci. Především pak se zaměřením na případné problémy při návrhu implementace těchto metod.

4.1.1 Agresivita

Cílem této metody je nahrazování známých přímých konstant v rámci optimalizovaného programu[3]. Tato metoda je implementována třídou *agresivity*, která je potomkem třídy *ForwardFlowAnalysis*. Tato třída je zodpovědná pouze za nalezení vyhovujících proměnných a sestavení data-flow hodnot pro jednotlivé bloky. Samotná úprava zdrojového kódu je pak implementována ve třídě *absorptionWrapper*, která poskytuje metody, jejichž návratovou hodnotou jsou data-flow hodnoty na vstupu, případně výstupu požadovaného bloku. Dále pak obsahuje metodu sloužící k optimalizaci zdrojového kódu na základě těchto hodnot. Jednotlivé data-flow hodnoty jsou reprezentovány pomocí dvouprvkového asociativního pole, kde jeho klíči jsou logické hodnoty true, případně false. Těmto klíčům pak přísluší množiny, obsahující proměnné, které jsou v daném bodě určitě pravdivé, respektive nepravdivé.

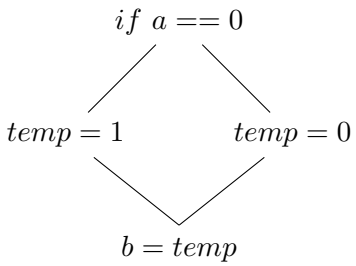
4.1.2 Negace

Při implementaci této metody, jsem předpokládal, že ve zdrojovém kódu neexistují vícenásobné negace. Proto je příslušná třída schopna nalézt, a správně identifikovat jen páry proměnných a jejich negací. Samotná analýza zdrojového kódu probíhá ve třídě *negs*. U této metody bylo, z důvodu vlastností Jimple reprezentace, nutné použít branched verzi analýzy. Rodičem třídy *negs* tedy je *ForwardBranchedFlowAnalysis*. Mimoto, jsem také vytvořil třídu *negsWrapper*, která obsahuje funkce sloužící ke čtení vstupních a výstupních hodnot pro jednotlivé bloky.

Jelikož neobsahuje Jimple reprezentace přímou instrukci, která by prováděla negaci daného výrazu, je negace prováděna za pomoci podmíněných a nepodmíněných skoků. Typickou ukázku takovéto posloupnosti příkazů demonstruje obrázek 4.1.

Samotné data-flow hodnoty reprezentuje asociativní pole, jehož klíči jsou jednotlivé proměnné a hodnotami pak množiny, obsahující všechny proměnné, jež jsou negací příslušného klíče v daném bodě. Toto pole je pak jedním ze dvou prvků dalšího asociativního pole. Druhým prvkem je pak pomocné pole se stejnou strukturou jako pole první. Toto druhé pole

[3]Podrobnosti je možno nalézt v sekci 3.4



Obrázek 4.1: Ukázka negace v Jimple reprezentaci

slouží pro uchování mezivýsledků při analýze kódu popsaného na obrázku 4.1. V jakémkoliv okamžiku obsahuje toto pomocné pole nejvýše dva prvky. Narazí-li analyzátor na podmíněný výraz, jež porovnává proměnnou a hodnotu 0, uloží si tuto proměnnou jako klíč do pomocného pole. Jako odpovídající hodnotu k tomuto klíči pak uloží hodnotu 1 respektive 0 pro pravdivého respektive nepravdivého následníka tohoto bloku. Obsahuje-li pomocné pole právě jeden prvek, pak analyzátor očekává, že následující příkaz bude přiřazení hodnoty určité proměnné, přičemž tato hodnota je rovna hodnotě odpovídající jedinému klíči v pomocném poli. V opačném případě je pomocné pole vyčištěno. Takto jsou analyzovány obě větve podmíněného výrazu. Prvním výrazem po sloučení těchto větví, by pak mělo být přiřazení proměnné z těchto větví proměnné třetí. Je-li tomu tak, pak jsou obě proměnné z pomocného pole a tato cílová proměnná přesunuty do pole výsledného. V případě že pomocné pole obsahuje nějaké prvky, ale analyzátor narazí na neočekávaný příkaz, je toto pole vyprázdněno až do doby než je nalezen další podmíněný výraz.

4.1.3 Absorpce

Tato analýza prochází zdrojový kód a vyhledává všechny výrazy používající operaci AND případně OR. Nalezne-li takovýto výraz, pak jej uloží do asociativního pole, jehož klíči jsou cílové proměnné jednotlivých výrazů. Struktura obsahující samotný výraz pak ukládá název cílové proměnné, jeho operandy a jednotku, ve které byl tento výraz definován. Dojde-li k šíření tohoto výrazu, to znamená, že nějaké proměnné je přiřazena hodnota, která je výsledkem některého z výrazů, jež se nachází v příslušném poli, je pak tato proměnná také přidána do tohoto pole. Operandy pro takovouto proměnnou jsou pak zkopírovány z původního výrazu. V závislosti na použité operaci (AND a OR) pak existují dvě samostatná pole reprezentující tyto operace. Třída zabezpečující tuto analýzu je nazvána *absorption* a je potomkem třídy *ForwardFlowAnalysis*.

Samotná optimalizace kódu pak probíhá v metodě *Optimize* třídy *absorptionWrapper*. Optimalizátor pak vyhledává výrazy používající operaci AND případně OR a porovnává operandy takovýchto instrukcí s hodnotami získanými ze třídy absorption. Při zkoumání negací v rámci těchto výrazů (viz tabulka 3.5 v sekci 3.6) pak slouží třída negsWrapper a stav množin obsahujících negace pro aktuální blok. Toto je bezpečné, jelikož při změně libovolné proměnné jsou jednotlivé množiny testovány a pokud některý z výrazů tuto proměnnou obsahuje, pak je z ní odstraněn.

Kapitola 5

Závěr

Cílem této práce bylo seznámení se s existujícími metodami optimalizace zdrojového kódu během jeho překladu a následné navržení metod vlastních. Tyto používané metody jsou představeny na začátku práce. Dále jsou zde představeny teoretické formalismy užívané při návrhu a popisu analýz a optimalizací zdrojového kódu. Při zkoumání existujících metod jsem zjistil, že tyto metody jsou zaměřeny na optimalizaci obecných výrazů a že by je tedy šlo upravit takovým způsobem, aby prováděli zjednodušení kódu na základě vlastností booleovy algebry.

Navrhl jsem tedy několik metod, zaměřených na zkoumání logických výrazů, zvláště pak jejich chování nabývá-li některý z operandů známé hodnoty. Druhá z navržených metod pak vychází ze způsobu chování výrazů, jejichž operandy, případně operandy některých jejich podvýrazů, jsou vzájemnou negací. Dále jsem se zaměřil na vztahy mezi složitějšími výrazy a jejich podvýrazy. Takovéto výrazy jsem se pak snažil transformovat na výrazy ekvivalentní, využívající menší množství instrukcí. Tyto optimalizace pak vedli k zavedení mrtvého kódu a tudíž i ke zmenšení optimalizovaného programu. Poslední z metod se pak snaží simulovat možné vstupní hodnoty jednotlivých výrazů a na základě takto získaných hodnot pak následně určit vzájemné vztahy mezi podvýrazy. U navržených metod jsem pak zkoumal, splňují-li vlastnosti, jako je například distributivita, které ovlivňují kvalitu řešení, jenž tyto analýzy poskytují.

Některé z navržených metod jsem následně implementoval. Mimo to jsem také vytvořil několik testovacích příkladu pro demonstrování funkčnosti těchto metod. Tyto ukázkové příklady je možno nalézt na přiloženém datovém nosiči.

Navržené metody nemají za cíl nahradit metody existující. Vzhledem k tomu, že často požadují, aby byl vstupní zdrojový zapsán specifickým způsobem, nemusí být jejich úspěšnost tak vysoká jako u metod obecných. Na druhou stranu dokážou, při vhodných podmínkách, zjednodušit i složitější výrazy, případně odhadnout výsledek výrazu i při znalosti jen jednoho z jeho operandů. Jelikož bývají logické výrazy často součástí podmíněných výrazů, může následně dojít k odstranění celých větví takovýchto výrazů a tím zmenšení výstupního zdrojového kódu.

5.1 Možná rozšíření

Kromě metod popsaných v této práci, je možno při optimalizaci využít i některých dalších vlastnosti booleovy algebry jako jsou De Morganovy zákony či distributivita. Nevýhodou těchto vlastností je, že jsou ještě více závisle na způsobu, jakým je zkoumaný zdrojový kód

zapsán.

Dalším možným využitím těchto metod, by mohla být jejich modifikace pro potřeby ladění programu. Analýzy by pak mohly programátora například upozornit, že podmíněný příkaz je tautologií, případně že výraz a jeho podvýraz jsou ekvivalentní, a že tudíž mohlo například dojít k překlepu během psaní kódu.

Literatura

[1] Domovská stránka projektu Soot. 2012, [Online; Navštíveno 14.3.2013].
URL http://www.sable.mcgill.ca/soot/

[2] Aho, A. V.; Lam, M. S.; Sethi, R.; aj.: *Compilers: Principles, Techniques, Tools – 2nd ed.* Pearson Education, 2007, ISBN 0-321-48681-1.

[3] Bartsh, H.-J.: *Matematické vzorce.* Academia, 2006, ISBN 80-200-1448-9.

[4] Bodden, E.: First steps using Soot 2.3.0 as a command-line tool. 2008, [Online; Navštíveno 10.5.2013].
URL http://www.bodden.de/2008/08/21/soot-command-line/

[5] Einarsson, A.; Nielsen, J. D.: A Survivor's Guide to Java Program Analysis with Soot. 2008, [Online; Navštíveno 16.3.2013].
URL http://www.brics.dk/SootGuide/sootsurvivorsguide.pdf

[6] Lam, P.: On the Soot menagerie – fundamental Soot objects. 2000, [Online; Navštíveno 16.3.2013].
URL http://www.sable.mcgill.ca/soot/tutorial/menagerie/menagerie.pdf

[7] Lam, P.: Using the Soot flow analysis framework. 2000, [Online; Navštíveno 16.3.2013].
URL http://www.sable.mcgill.ca/soot/tutorial/analysis/analysis.pdf

[8] Muchnick, S. S.: *Advanced compiler design implementation.* Morgan Kaufmann Publishers Inc., 1997, ISBN 1-55860-320-4.

Příloha A

Obsah CD

Přiložený datový nosič obsahuje:

- Zdrojové kódy implementovaných metod, včetně knihoven nutných k jejich sestavení.
- Text bakalářské práce ve formátu PDF.
- Zdrojové kódy pro překlad textu bakalářské práce.
- Sadu vzorových příkladů pro testování implementovaných metod.

Příloha B

Manuál

Součástí přiloženého nosiče je složka *implementation*. V této složce se nalézají soubory jasminclasses-2.3.0.jar, polyglotclasses-1.3.5.jar a sootclasses-2.3.0.jar. Tyto soubory jsou převzaty z webové adresy `http://www.sable.mcgill.ca/soot/soot_download.html`, obsahují implementaci samotného frameworku Soot a jsou nutné ke spuštění a překladu jednotlivých optimalizací. Dále tato složka obsahuje soubor makefile, sloužící k překladu[1] a spuštění přiložených analýz. Tento makefile definuje příkazy *make compile* (volán i pokud je volán make bez parametrů), sloužící k překladu přiložených zdrojových kódů, dále příkaz *make run*, sloužící k provedení analýz[2] na ukázkových příkladech a příkaz *make clean* sloužící k odstranění přeložených souborů a výstupů jednotlivých analýz. Složka *implementation/sootOutput* obsahuje podsložky *after* a *before*. Při zavolání příkazu make run je nejprve zavolán framework Soot bez mnou implementovaných metod, při tomto zpracování jsou ukázkové třídy převedeny do jimple reprezentace a uloženy do složky before. Následně je Soot zavolán znova, tentokrát ovšem i s implementovanými metodami, výsledek tohoto druhého průchodu je pak, opět v jimple reprezentaci, uložen do složky after. Při optimalizacích nevyužívám žádné optimalizace sloužící k odstranění mrtvého kódu. Výstupní soubory tedy obsahují i instrukce, které by tato skupina metod odstranila. Zdrojové kódy tříd popsaných v této práci se pak nalézají ve složce *implementation/src*. Třídy určené k optimalizaci, sloužící k demonstraci funkčnosti zde popsaných analýz, se pak nalézají ve složce *implementation/sootInput*.

[1] Testováno na školním serveru Merlin.

[2] Návod pro ovládání frameworku Soot z příkazové řádky je možno nalézt na [4].